Designing real-time dependable distributed systems

Gerard Le Lann

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P 105
78153 Le Chesnay Cedex
France
Tél. (1) 39 63 55 11

Rapports de Recherche

1 9 9 2

25ème anniversaire

N° 1425

Programme 1
Architectures parallèles, Bases de données,
Réseaux et Systèmes distribués

DESIGNING REAL-TIME DEPENDABLE DISTRIBUTED SYSTEMS

Gérard LE LANN

Avril 1991

RR-1425

LA CONCEPTION DE SYSTEMES TEMPS REEL REPARTIS SURS DE FONCTIONNEMENT

Gérard Le Lann
INRIA, Projet Reflecs, BP 105
78153 Le Chesnay Cedex, France

gll@score.inria.fr

Résumé :

Les études portant sur les concepts de répartition, de temps réel et de sûreté ont été menées plus ou moins séparément jusqu'à une date récente. Cet article a pour but de présenter ces concepts, de donner des définitions rigoureuses et de tenter de clarifier certains problèmes de conception majeurs posés par les systèmes informatiques temps réel répartis sûrs de fonctionnement. En particulier, certaines erreurs classiques sont analysées ainsi que la relation qui existe entre la "prédictabilité" d'un système et la complexité des hypothèses à la conception.

Dans cet article, on examine également les relations existant entre les trois domaines du temps réel, de la répartition et de la sûreté, en portant une attention particulière au problème de la compatibilité algorithmique, dont l'importance n'est pas encore très bien perçue.

DESIGNING REAL-TIME DEPENDABLE DISTRIBUTED SYSTEMS

Abstract :

The concepts of distribution, real-time and dependability have been investigated more or less separately until recently. This article reviews these concepts and, based on rigorous definitions, attempts to clarify some of the most important design issues raised with real-time dependable distributed computing systems. In particular, some popular misconceptions are examined as well as the relationship existing between predictability and design assumption complexity.

This article also investigates the relationships existing between the three areas of distribution, real-time, dependability and focuses on the issue of algorithmic compatibility, whose importance has not been fully acknowledged yet.

CONTENTS

1. INTRODUCTION

The concepts of distribution, real-time and dependability have been investigated more or less separately until recently. Although each of these concepts is reasonably well mastered in its own context, a good understanding of those relationships existing between the three corresponding areas has not been fully developed yet. This article is intended to help establishing a better identification of the issues raised with the design of real-time dependable distributed computing systems.

Real-time is a concept viewed as matching the properties of timeliness or punctuality. The concept of distribution is related to the notion of asynchronous parallelism, as found in physically dispersed architectures supporting concurrent computations. Finally, dependable systems should behave predictably. Consequently, for the sake of simplicity, real-time dependable distributed systems will be referred to as P* systems in this article, in that they are or should be punctual, parallel and predictable (they could, in addition to the above, be persistent, polymorphous, prompt, etc.). In section 2, concepts and definitions are given. Some popular misconceptions are presented and it is shown that current commercial offerings have limited scope of applicability with respect to P* system properties. In section 3, the two major approaches to the problem of designing highly predictable P* systems are reviewed and compared. Section 4 addresses the issue of algorithmic compatibility, that ensues from the need to blend together those algorithms needed to achieve the properties sought.

2. CONCEPTS, DEFINITIONS AND MISCONCEPTIONS

Many existing commercial offerings - operating systems in particular - are advertized as being entrusted with distribution and/or real-time capabilities with some degree of dependability. However, most often, analysis reveals that such offerings do not meet well accepted definitions that have been elaborated years ago. Consequently, users may learn the hard way that their computerized applications do not behave as expected.

Examples of mistaken views abound. For instance, the discrimination between centralization and distribution is based on whether or not communication protocols are used. The discrimination between real-time and non real-time is based on magnitude of response times. The discrimination between fault-tolerance and non fault-tolerance rests on whether or not back-ups are used.

2.1. Definitions

2.1.1. Distribution

There is a need to discriminate between **physical dispersion** (as found in existing computer-communication networks) and **global control of concurrent computations** (as found in a few commercial multiprocessor offerings today). Pioneering work in this area[1,2,3] was conducted in the mid 70's.

The distinctive attribute for any given computing activity is whether or not a unique (centralizing) entity is relied upon to conduct that activity. This distinction between centralization and distribution is independent of underlying physical architectures. For example, a computer network could be used to implement a centralized system, whereas a shared-memory multiprocessor could be designed as a distributed system.

Definition of a distributed system

A computing system whose behavior is determined by algorithms explicitly designed to work with multiple loci of control and aimed at handling concurrent asynchronous computations.

Examples of such algorithms are concurrency control schemes found in distributed On-Line Transactional Processing systems. We will say that a system is entrusted with distributivity properties to mean that its design is an accordance with the definition given above.

2.1.2. Real-time

There is a need to express the value gained by a system (resp. the loss incurred) whenever a thread of computation terminates "in time" (resp. "too late"). Such values or losses (i.e. negative values) may be expressed as functions of physical time[4].

The following definition is a generalization of more conventional definitions of real-time that, in general, fail to capture what system behavior is in case schedulability conditions are not satisfied.

Definition of a real-time system

A computing system where initiation and termination of activities are explicitly managed to meet specified timing constraints. Time-dependent values are associated with activity terminations. System behavior is determined by algorithms designed to maximize a global time-dependent value function.

Simple examples of such algorithms are Earliest Deadline First and Shortest Slack Time First.

A system designed in accordance with this definition is said to be entrusted with timeliness properties.

2.1.3. Dependability

There is a need to express the fact that a system behaves as specified even in the presence of such "hostile" actions as fault occurrence or intrusions.

In other words, and following the terminology established by IFIP W.G.10.4, a dependable system is such that the occurrence of some undesired actions (e.g. faults) does not necessarily lead to a system failure.

Definition of a dependable system

A computing system whose behavior is determined by algorithms explicitly designed to cope with some given number of simultaneous "hostile" events, for given event classes.

Examples of dependability measures are availability, security or safety[5]. Examples of techniques used to obtain quantifiable dependability are fault-tolerance and cryptography.

A system designed in accordance with this definition is said to exhibit dependability properties.

2.2. Misconceptions

There are a number of biaised views or misconceptions related to the design of P* systems. In the following, we elaborate on a few of them.

2.2.1. Distribution versus networking

Most existing multicomputer systems and many multiprocessor systems where message-passing protocols are used to offer interprocess communication are marketed as being distributed. This is the case in particular with those systems that implement some ISO/OSI protocol stack, or anything equivalent (e.g. TCP/IP, NFS, X400).

All existing ISO/OSI or other communication protocol standards have been developed for one specific target : general purpose interworking in heterogeneous environments. Sliding window based end-to-end message transfer or client-server oriented services (e.g. RPCs, MMS, ROS) only allow for reliable explicit interaction among two processes, that are mutually aware of their respective existence.

This has nothing to do with the issue of distribution, as defined previously. Even with sophisticated ISO/OSI-like communication protocols, orderings of message deliveries among connections established by application processes residing over multiple computers/processors are arbitrary. In general, these orderings are determined by some multiplexing protocols used to grant access to shared resources (e.g. communication channels, network interface units, local executives, etc.). No global synchronization algorithm being used (e.g. as those based on a distributed timestamping scheme), it is obvious that the resulting set of "assembled" orderings has no particular meaning and therefore no property attached to it. For example, the consistency of updatable distributed files could be destroyed. Message-passing in heterogeneous systems is not equivalent to distribution.

2.2.2. Distribution versus portability

Good system software engineering practice has recently made inroads in the commercial operating systems arena. Operating systems whose architecture obeys the well known principles of modularity and decomposition are viewed nowadays as being quite advanced. Offerings from the Open Software Foundation, Unix International and a few free-wheelers are/will be based on a kernel oriented approach.

A kernel includes some code that is hardware-dependent. It is claimed that the servers or the operating system processes that run on top of a kernel are hardware-independent.

Although such an approach should ease to a large extent the task of reconfiguration and porting of operating systems over various hardware architectures, it is by no means related to the concept of distribution as defined above. Indeed, existing OS offerings, so-called "distributed", have been designed assuming shared-memory architectures or they include conventional communication protocols (see §.2.2.1). Kernel-based designs or portability are not equivalent to distribution.

2.2.3. Real-time and response times

In some circles, real-time means short response times or high sampling rates. In other circles, real-time means short interrupt handling latency or fast task context switch. Of course, there is total disagreement in those circles as whether 10 ms or 1s should be a "good" response time value, or 2 ms rather than 20 ms should be a "good" context switch latency value, so as to discriminate between "real-time" and "non real-time" systems. There are three major misconceptions in the area of real-time scheduling, presented below.

Real-time computing is fast computing

Let us consider two computing systems, one called the Tortoise and the other called the Hare*. The Tortoise has the following features :

- speed : 1
- task scheduling and context-switch latency : 1
- scheduling policy : earliest deadline first, no preemption.

The Hare has the following features :

- speed : 10
- task scheduling and context-switch latency : 0
- scheduling policy : first-come-first-served, no preemption.

Gain ratios of the Hare over the Tortoise therefore are :

- 10 in terms of raw processing power
- ∞ in terms of task scheduling/switching latency.

In some circles, the Hare is obviously "more" real-time than the Tortoise.

Let us now consider an application comprising two tasks, and the following situation, typical of real operational conditions. At time t, task A is pending, about to be scheduled and run. However, also at time t, a request for activating task B is triggered. Tasks A and B are equally critical tasks.

Let us assume that task attributes are as follows :

Task A : duration : 270, at speed 1,
deadline : t + 320

Task B : duration : 15, at speed 1,
deadline : t + 21

Execution patterns would be as shown figure 1, for the two systems considered.

* after the tale "The Hare and the Tortoise", from the French writer Jean de la Fontaine (1621-1695)

Conclusion is obvious. The Tortoise, which is slow, but which processes tasks according to a correct sequence, meets both deadlines. This is a safe real-time system for the application considered. The Hare, which runs tasks very fast, but in the wrong order, does not meet both deadlines. The Hare is definitely unsafe for the application considered.
Real-time computing is not equivalent to fast computing. It is of course possible to build counter-examples, showing that the Hare can win against the Tortoise. However, such examples would do no more than demonstrating that with a brute force approach (over-dimensioned systems), one can do as well as with a clever approach and correctly dimensioned systems.

System over-dimensioning, not a solution, rather demonstrates refusal to look for a solution. Furthermore, there are cases where over-dimensioning is antagonistic with weight and energy requirements (e.g. space applications).

Fast computing might help but does not suffice. Appropriate scheduling algorithms are necessary.

Real-time computing is fast context switch

In the example given above, the Hare system, which was assumed to have a zero time context switch, loses against the Tortoise for the application considered. What if preemption would be allowed ?

Many commercial systems are coined "real-time" because they efficiently handle hardware generated interrupts and do task context switching very rapidly. It suffices to compute the savings induced with small preemption latency (time to switch task contexts) to show that fast context switch is not germane to real-time computing.

For any "reasonably well" constructed system, p, the average preemption latency, and D, the average task duration, should be such that $p/D << 1$.

Worst-case corresponds to experiencing a succession of n preemptions before running the highest priority task, if n+1 priority classes are used.

The savings achieved by an ideal "real-time" system (p=0) compared to a "non real-time" system would then be measured by the ratio $1/(1+np/D)$.

Firstly, let us observe that any system running at a speed $1/(1-np/D)$ times higher than the ideal system would perform as well. For example, with D=50 ms and n=15, a system such that p=50 microseconds needs to run approximately 1.5 % faster than the ideal system to perform as well.

Secondly, the ultra fast preemption argument misses the real issue again. Indeed, in order to achieve some impressively small preemption latency, scheduling algorithms implemented within such "real-time" systems usually are very primitive (yielding fast scheduling decisions).

For example, scheduling schemes used in most current commercial systems make use of fixed priorities such as hardware interrupts. There is a major problem with these schemes when not used in their intended context (see further). Basically, fixed priorities are alien to the notion of time. Those time dependent attributes that are associated with application tasks in real-time systems are simply ignored.

Therefore, it does no matter so much whether wrong scheduling decisions or, at best, poor scheduling decisions, are made quickly or not. What matters is that such fast preemption based systems do not make the right decisions. Using the example given above again, one obviously sees that if B is (a priori) attributed a priority level inferior to A's level, preemption does not help.

Furthermore, the ultra fast preemption argument demonstrates a confusion of ends and means, in that it is based on the belief that starting a process as fast as possible is equivalent to guaranteeing timely completion of that process and of other processes it is contending with.

The identification of an appropriate scheduling algorithm is the key issue. Raw hardware performance is of secondary importance.

Fixed priorities guarantee timeliness

Most existing commercial "real-time" systems or operating systems schedule processes according to a hierarchy of priority levels. At first glance, such systems seem simple and easy to use. The intuitive idea is that it suffices to assign a priority level (an integer) to every process prior to letting a system run. This is called the priority mapping function. At run time, the scheduler always selects the pending process with the highest priority level.

Of course, in real-time systems, application processes have timing constraints attached to them (e.g. deadlines, frequencies, time-value functions). As indicated previously, for general assumption sets (e.g. aperiodic arrivals, time-value functions), corresponding scheduling problems are known to be NP-complete. And so is the priority mapping problem.

Hence the questions : How can time dependent constraints be rigorously translated into time independent integers ? Is this translation time-independent ? How can it be suggested that the priority mapping problem can be easily "solved" by users ? How to guarantee that fixed priority based scheduling does not yield starvation and probabilistic service for all processes except those mapped onto the highest priority level ?

For particular cases, the priority mapping problem can be rigorously solved. For example, the rate monotonic theory[6] and its more recent variants[7] can be used to compute process priorities off line and to check a priori whether sufficient conditions are met for schedulability. However, such approaches correspond to restrictive assumptions which, unfortunately, do not match well the characteristics of distributed systems or dependable systems. Therefore, proofs established for "simple" cases (e.g. uniprocessor-like systems, no unanticipated resource conflict, static partitioning of the resource set, etc.) have restricted applicability. An interesting breakthrough in this area would consist in establishing sufficient conditions for schedulability in distributed systems and derive provably correct scheduling algorithms that would use time-dependent attributes.

3. ASSUMPTIONS, PREDICTIONS AND ILLUSIONS

3.1. Predictability

A P* system must behave as prescribed by its specifications, i.e. it must be predictable in the logical domain (values) and in the physical domain (timings). Ideally, specifications should be complete, i.e. they should fully encompass every future run-time condition.

When not the case, a P* system should be designed in such a way that behavior corresponding to unexpected conditions still is predictably correct.

Consequently, predictability measures the likelihood that :

(i) assumptions made at specification/design time are not violated at run-time and,

(ii) system behaves as anticipated whenever run-time conditions match specification/design assumptions.

The design and implementation of P* systems is rendered difficult by the existence of the following physical facts that characterize real world systems and environments :

- multiplicity of asynchronous hardware elements
- occurrence of faults and intrusions
- finite space, speed and energy levels
- passing of time.

As a consequence (but it is only a consequence of those basic facts), in the general case, delays for computing, for communicating, cannot be known or predicted with certainty. This is a major impediment to the realization of P* systems. Furthermore, in general, system loads are variable and cannot be fully anticipated, this being particularly true with distributed systems. Consequently, all problems derive from the following basic dilemma : how to build a predictable system when (i) the system environment (ii) the system constituents, exhibit non fully predictable behaviors ?

3.2. Assumptions and computational models

There are basically two major approaches to the predictability issue. One is based on the idea that future system behavior can be fully "guaranteed". The other is based on the idea that absolute predictability is not achievable.

Future system behavior depends on what can be called future circumstances. Examples of future circumstances that must be fully and rigorously predicted or characterized are listed below.

(i) Environment-wise

* input arrivals laws
* relative/absolute timings of input arrivals
* characterization of criticality for each input arrival, i.e. its importance/value with respect to global system mission :

    - in a time-dependent or time-independent manner ?
    - dependent on current global system state or not ?

(ii) Environment-wise and system-wise

* fault "arrival" laws, per fault class
* relative/absolute timings of fault arrivals

* optimal/sub-optimal system reconfiguration patterns according to fault occurrence and global system state.

Furthermore, issues raised by the computational model used must also be addressed rigorously.

Computational model issues

Let us consider that an input arrival results into the activation of one or several tasks. Examples of assumptions directly related to the type of computational model considered are as follows :

* sequential or concurrent tasks ?

* can intertask conflicts occur ? If so, what are the possible conflicting patterns ? Are the relative/absolute timings of internal events (e.g. message-passing) supposed to be known ?

* time-bounded or time-unbounded computational/communication delays ? If a time-bounded delay model is used, how is it guaranteed that timing failures never occur or are always detected ?

* how much "guaranteed" is the precision of the global timing used (synchronized clocks) ?

Clearly, most often, these future circumstances cannot be predicted with some sufficient degree of confidence.

3.3. Complexity and predictability

Let us define a, the application intrinsic complexity (that includes the environment), as a variable that can be unambiguously valued (from 0 to ∞) for any given application.

A good specification/design is such that it is an accurate model of the application of interest. Let us define d, the specification/design assumptions complexity, as a variable that can range from 0 to a, for any given application.

Let us now define α, the assumption coverage factor, as the ratio d/a. Ratio α could be viewed as the probability that specification/design assumptions are not violated at run-time.

Let us assume that some method is used to prove/verify that a design is correct (i.e. specifications are satisfied). Let γ be the verification/proof coverage factor, i.e. the fraction of the system design that has been verified/proved correct. Ratio γ could also be viewed as the probability of entering system states at run-time that have been verified/proved correct.

Predictability P is then defined as the product $\alpha\gamma$.

Let us now consider some application of given complexity a. Let us consider two possible designs. One is qualified as being ε-complex, to indicate that it carries little complexity

(d << a). The other is qualified as being s-complex, to indicate that it is satisfactorily complex (d ≅ a).

For a modern avionics application, an example of an ε-complex design would be as follows :

- fully deterministic assumptions (e.g. upper bounds for input arrival frequencies, such as sensor reads)

- time-independent task criticality

- uniprocessor-like architecture, no task interference

- fully deterministic bounds on fault occurrence.

A s-complex design in the same application area would correspond to taking into account the requirements defined for the US Air Force Pave Pace mission management or for the Pilot's Associate.

On figure 2, it is shown that P is not necessarily higher with ε-complex designs. Furthermore, it is highly likely that in the near future (i) critical applications of significant complexity will be computerized, and (ii) new verification/proof methods will be established that encompass higher levels of complexity. Hence ε-complex designs such as those used today should be departed from.

3.4. Illusions

It is quite clear that the concept of absolute predictability is inherently flawed. Such a concept derives from the belief that devising a design such that γ=1 suffices to justifiably claim "guaranteed" system behavior. This is the first illusion.

Such designs, that are inevitably simple, if not simplistic, illustrate what could be called a static approach.

A static approach is based on :

- full enumeration of every possible future circumstance
- exhaustive (deterministic) a priori verification of properties and behavior for all possible system states.

In this context, probabilistic verification is not viewed as being appropriate, in that it does not yield deterministic "guarantees".

An example of a static approach in the area of real-time task scheduling would be table-driven scheduling. However, static system behavior is not predictable, as being unspecified, for the set of unanticipated circumstances. At best, one could imagine static designs such that a system is mute whenever facing some unanticipated circumstance. Is this the desired behavior, always ?

Conversely, what could be called a dynamic approach corresponds to devising a set of assumptions such that α is sufficiently close to 1. Most often, this results into a design that includes "dynamic" algorithms, i.e. algorithms not resting on a priori computed scenarios. An example of a dynamic approach in the area of multiaccess protocols would

be carrier-sense multiple access/collision detection (CSMA-CD), in contrast with token-passing or polling (see further).

The second illusion is that designs based on dynamic algorithms cannot be verified. Dynamic systems can obviously be verified under assumption sets identical to those considered with static systems. However, dynamic systems keep running even in the face of unanticipated circumstances. In other words, they are mute only if so specified. For all other cases, their behavior is derived from "interpolation" between those behaviors specified for circumstances closest to the one encountered. Dynamic system behavior is predictable over regions of a given assumption set, not just over a collection of elements in that set, as is the case with static systems.

Let us illustrate the superiority of a dynamic approach over a static one with the multiaccess problem.

3.5. An example

For both slotted or unslotted shared communication channels, one basic question that arises is whether self-adaptive protocols (e.g. ISO/OSI 8802/3 or CSMA-CD) are better or worse than static protocols (e.g. static TDMA).

When such shared channels are used within P* systems, their access must be controlled by a protocol that guarantees distributivity, timeliness and dependability properties.

Distributivity and dependability requirements (e.g. availability) usually preclude the use of token-passing protocols, such as token-bus (ISO/OSI 8802/4) and token-ring (ISO/OSI 8802/5, FDDI). This is because it is generally very difficult to predict the probability of occurrence of token losses. Contrary to statements that can be found in the Manufacturing Automation Protocol documentation, on-site experience has demonstrated that token losses can occur at high rates with current implementations of token-bus based offerings.

The ISO/OSI 8802/5 token passing scheme is also vulnerable in that the access control field that contains the token bit (and the priority/reservation bits) is not covered by the frame check sequence. The token bit may thus be changed (i.e. the token may be lost) without this being detected by ring attachment units.

Both protocols provide for token regeneration. Nevertheless, token losses can induce damaging effects for application level software. For example, a communication blackout of 100-200 ms at bus/ring level might result into a communication blackout of several seconds at user level.

Conversely, non token-passing protocols, such as the CSMA-CD protocols used in Ethernet and in more recent offerings, exhibit excellent availability properties in the presence of noise. Spurious disturbance is simply handled as a collision, that constitutes a normal event for CSMA-CD channels.

Timeliness is usually considered as being achievable only with either token-passing protocols or static TDMA. This mistaken view is based on the fallacy that token-passing is inherently "deterministic". Elementary analysis reveals, quite obviously, that "determinism" rests on full a priori knowledge of message arrival laws, reconfiguration timings (network attachment units going down or joining in) and token loss/token regeneration timings.

Clearly, assuming such a priori knowledge is unrealistic. Consequently, token-passing protocols are not more "deterministic" than other protocols. More correctly stated, the predictability of token-passing protocols is usually smaller than that of contention protocols, except for a few particular cases.

Algorithmically speaking, the ISO/OSI 8802/4 standard probably is the weakest of all existing standardized token-passing protocols, in that it is based on a fundamental design contradiction. This protocol derives from the "axiom" that collisions should be ruled out (hence the token). In fact collisions can occur (when the token is lost, when repaired/new stations join in the virtual ring). Consequently, the token-passing bus standard includes specific procedures meant to cope with collisions. This strange design exercise has an analogy in Mathematics : demonstration of a theorem that violates one of the original axioms.

What matters is whether or not time-constrained messages are transmitted "in time" over a multiaccess channel. Let us abstract every network attachment unit as a waiting queue of messages. Every message is associated a time attribute, e.g. a deadline. Messages are ranked in each waiting queue according to some universal criterion, e.g. by increasing deadline values. This is a good model for P* systems, in that is allows multiaccess protocols to operate in accordance with attributes directly derived from those timing constraints expressed at the application level.

Competition takes place among those messages at the head of each of the waiting queues. It is indeed possible to use contention protocols to build "deterministic" multiaccess channels. For example, with the ISO/OSI 8802/3 protocol, if binary tree search would supersede the binary exponential backoff algorithm, the resulting protocol would yield guaranteed upper bounds on collision resolution latency, for every possible situation (including worst-case, i.e. general collision). Deterministic collision resolution could be based on names of attachment units or based on slack times (deadline - current time) and names in case of identical slack times.

Deterministic CSMA-CD is an example of a protocol that belongs to the particularly interesting class of tree protocols. Tree protocols have been thoroughfully analyzed[8,9]. Contrary to token-passing, exact analytical models have been established. Tree protocols enjoy the channel transparency property, they exhibit excellent stability thresholds and their deterministic variants outperform token-passing protocols for a vast majority of assumption sets corresponding to realistic situations. Tree protocols also exist for high-speed unidirectional channels, with probabilitic and deterministic variants.

Compared with token-passing schemes, that are based on fixed priority scheduling, deterministic CSMA-CD does not raise the issue of priority mapping (see section 2.2.3). Deterministic CSMA-CD is obviously superior in terms of availability properties and recovery latency (in case of channel jamming). A coupling with global time-dependent scheduling algorithms is feasible with deterministic CSMA-CD, whereas it is unfeasible or impractical with token-passing, being defeated by the utilization of fixed priorities at the multiaccess layer. Such a coupling clearly brings valuable advantages with respect to timeliness. For example, the coupling of deterministic CSMA-CD with EDF scheduling yields a powerful multiaccess scheme in the sense that all message sets that meet those sufficient schedulability conditions required by fixed-priority based scheduling algorithms are schedulable with EDF.

3.6. The power of proofs

Of course, exhaustive state space exploration is not the only approach to obtaining some degree of predictability. Proofs can be used. In fact, it is believed that proof-based approaches only can be considered whenever application intrinsic complexity is significant and required predictability is high (e.g. 1 - 1.10^{-10}). State-of-the-art of proof-based approaches is more advanced in some areas than in others. For example, let us consider achievements in the areas of concurrency control and task scheduling.

The serializability theory has provided the formal basis needed for establishing very powerful results[10]. For example, the problem of deciding whether an interleaved schedule of actions belonging to concurrently executing transactions is serializable or not is NP-complete. This problem used to be tackled either via prevention oriented approaches (global semaphore, no concurrent read-write) or via static approaches (restrictive assumptions on internal transaction structures, on resource-transaction relationships and on resource set partitioning). Since the mid 70's, proofs (theorems and sufficient conditions) for serializability have been and are being established, some of them meeting every possible pattern of concurrent executions (see[11] for an early example). A large number of concurrency control algorithms have been devised, in accordance with these proofs, making it possible to obtain distributivity properties with absolute predictability (α=1, γ=1). Examples would be MultiVersion Timestamp Ordering (MVTO) if one is interested in safety properties only or 2-phase locking + TO-based deadlock prevention if liveness properties are required as well.

It turns out that the scheduling theory has not generated equally powerful results yet. In the general case, where task timing constraints are expressed as time-value functions, the optimal scheduling problem is also NP-complete. Proofs for schedulability have been established for assumption sets that are still restrictive compared to real world P* systems. However, significant progress has been accomplished since the early 70's, when periodical arrivals/conflict free tasking models only were tractable. There is a growing interest in this area, as witnessed by the proofs and formal results established recently.

From this discussion, one can derive the following conclusions relative to P* system design issues :

(i) ratio α is determined by the smallest of those variables d that each represent the specification/design assumption complexity mastered for some given property of interest (e.g. timeliness, security)

(ii) for a given set of properties sought, one needs to know whether some combination of algorithms is optimal or even acceptable.

This is what we call the algorithmic compatibility issue, which is not being very much addressed. In fact, the importance of this issue has not been fully acknowledged yet.

4. THE ALGORITHMIC COMPATIBILITY ISSUE

4.1. Essence of the problem

Dependability properties are obtained out of redundancy (state data, hardware, software). Obviously, redundancy should be managed via distributed algorithms (no central point of vulnerability). Hence the logical relationship between dependability and distributivity. Timeliness properties cannot be ascertained if fault occurrence or denial of service could

impede progress for arbitrary durations. Hence the logical relationship between timeliness and dependability. Parallelism and multiprocessing are well-known approaches to significantly reducing computational delays. Hence the relationship between distributivity and timeliness.

Our goal is to show that the algorithmic aspects involved with designing P* systems do not boil down to selecting an algorithm that is deemed appropriate for each property of interest. Indeed, those "appropriate" algorithms will ultimately be integrated within a kernel or an operating system or some application-level software especially designed to run on a P* system. These algorithms must "cooperate" rather than defeat each other, as would be the case if they are incompatible.

It is worth noticing that the algorithmic compatibility issue is also raised when applications that run on pre-existing operating systems/kernels must be supplemented with those appropriate algorithms needed to achieve properties not provided by the underlying operating systems/kernels. Application-level designers must be aware of the fact that their spectrum of algorithmic choices is directly determined by those algorithms that have been implemented within the operating system/kernel chosen. In some cases, algorithmic incompatibilities are severe enough that properties of interest can only be obtained by circumventing the operating system/kernel chosen ("short-circuiting" it, in some sense) or by supplementing it directly with the appropriate algorithms. A classic example in the traditional real-time systems arena are the incompatibilities existing between the Ada tasking model on the one hand, with its implicit FIFO and non-deterministic servicing policies, and process scheduling algorithms required to satisfy specific application-level timeliness requirements on the other hand.

In the following, we give examples of algorithmic incompatibilities that may not be obvious at first glance. Such incompatibilities have far-reaching effects on how to correctly design P* systems.

4.2. Distributivity and timeliness

The first area of Computer Science to directly address the algorithmic aspects of distribution was that of distributed databases[12]. The issue of maintaining the consistency of a set of distributed data structures that can be concurrently created, destroyed, read, updated by a possibly unknown number of processes has been at the origin of fundamental results and concepts such as that of serializability and atomic transaction.

Most synchronization algorithms currently used in distributed multicomputer/ multiprocessor systems are refinements of those solutions that were devised to enforce the atomicity property for concurrently executing distributed transactions. Examples of these early algorithms are two-phase locking[11], timestamp ordering[13] and ticket ordering[14]. Atomic transactions with replicated databases were shown to be feasible with quorum protocols[15]. Since then, many solutions to the concurrency control problem have been published. In[16], existing concurrency control algorithms are categorized in three classes, namely locking protocols, non-locking schemes and multiversion concurrency control. Many of these algorithms resort to blocking or rolling back or aborting transactions in case the desired consistency properties would be endangered.

Consider now that timeliness is required, in addition to distributivity. Many existing concurrency control algorithms cannot be used.

Firstly, blocking or rolling back or aborting processes is intuitively antagonistic with the achievement of timeliness properties. Even worse, such schemes yield possibly largely varying process execution delays, ranging from "ideal" delays (no conflict) to "worst" delays (all possible conflicts occur), assuming a bound exists on the number of waits or rollbacks that a process can experience. It is thus clear that the merits of those scheduling algorithms proved to be optimal or acceptable assuming process durations can be accurately predicted may well need to be reassessed in the case of distributed systems.

Secondly, with real-time systems, intermediate states (e.g. visible outputs) produced while processes execute must be triggered in time, i.e. possibly before process termination, and cannot be cancelled later on. This might be antagonistic with rollback-based or abort-based algorithms.

To summarize, the algorithmic compatibility issue raised with combining distributivity and timeliness properties can be simply described as a problem of searching for schedule sets with a non-empty intersection.

Consider a system where a number of processes are running concurrently. Each process is viewed as a set of read/write actions. A concurrency control algorithm serves the purpose of dynamically filtering out those interleaved action schedules that would threaten distributivity properties. Let S_d be the subset of legal schedules, i.e. those that enforce such properties. Similarly, a real-time scheduling algorithm serves the purpose of dynamically instantiating those interleaved action schedules that satisfy timeliness requirements. Let S_t be the corresponding timely schedule subset.

Arbitrary choices of algorithms might result into having $S_d \cap S_t = \Phi$ (empty set). For example, a combination of the shortest-slack-time-first algorithm and a locking algorithm + deadlock detection/recovery based on process names might result into a mute system. Similarly, a combination of fixed priority scheduling with multiversion timestamp ordering is not acceptable.

Conversely, with those process models where a process timing constraint is expressed as a deadline, and where an action inherits the deadline associated with the process it belongs to, the earliest-deadline-first algorithm (EDF) fully matches a 2-phase locking algorithm + deadline-based deadlock avoidance (2PLDDA). In case of conflicts, legal schedules and timely schedules are strictly equivalent.

With other process models, e.g. where actions that belong to some given process may have different timing constraints, more elaborated solutions are required.

For practical purposes, deciding whether or not to use the EDF-2PLDDA combination, or any other compatible algorithmic pairing, is primarily determined by the merits of the real-time scheduling algorithm selected.

In other words, current state-of-the-art is such that the predictability of a distributed real-time system is almost always bounded by the predictability of the real-time scheduling algorithm considered.

4.3. Security and distributivity

A number of application areas where P* systems only can be considered raise questions of multilevel security, that is identification and separation of information and users based

on classification and clearance. Although of little or no concern at all until recently, security issues are now considered sensitive in many civilian applications, such as financial trading and factory/process control (following the trend in defense-oriented applications).

Consider a distributed file/database system that should be endowed with multilevel security properties. It has been shown that consistency - a distributivity property - and security - a dependability property - might be conflicting requirements.

This is the case for example with multilevel database systems layered on a trusted computing base and where single-level subjects only are allowed. It has been shown that standard 2-phase locking, timestamp ordering and MVTO algorithms may not be compatible with common security policies[17,18].

Lock/unlock actions (with 2PL) and timestamp modifications (with TO) could be used as a covert signaling channel, when performed on behalf of transactions that run at different levels and that happen to conflict with each other. Similarly, MVTO might yield rejection of a write action, in case that write would invalidate a value previously returned by a read action. Such rejections could also be used as a covert signaling channel.

Denial of service is another possible undesired side-effect of lock-based or timestamp-based concurrency control algorithms.

The various approaches followed to solve this algorithmic compatibility problem, whether based on trusted subjects or not, entail more or less significant modifications of previously known algorithms.

4.4. Fault-tolerance versus timeliness and distributivity

Fault-tolerance is of particular relevance with P* systems for fault-tolerance has to do with comparing the actual behavior (of a system, of a constituent) with some intended behavior. This implies that the observee and the observer are distinct entities, with independent fault modes if possible. Existence of multiple entities precisely is a pre-requisite for distributed systems. This issue is also of direct concern to designers of real-time systems. Indeed, the best way to violate timeliness requirements consists in paying no attention to the occurrence of faults. Conversely, time is the only means whereby one can tell whether an entity is faulty or whether it behaves correctly but slowly.

Achieving fault-tolerance in a distributed system goes beyond allowing a process x to detect that the behavior of some other process y is abnormal and letting x kill y. Process x could be the faulty process. Therefore, fault-tolerance in a distributed system implies the possibility for multiple entities to reach consensus (e.g. whether to passivate/activate an entity or a replicate). Reliance on one single entity would be in contradiction with the general principles of distributed computing. See[19] for a detailed presentation of those issues raised with fault-tolerance in a distributed system.

It has been shown that distributed consensus cannot be guaranteed in a time-unbounded system (often called asynchronous) in the presence of crash failures[20]. Under a deterministic approach, terminating consensus is an obvious prerequisite to the achievement of timeliness properties. Consequently, timeliness and distributivity appear to be contradictory with the need to tolerate faults, in the context of time-unbounded models.

The problem of how to obtain terminating or time-bounded distributed consensus has been investigated so far under two approaches, namely the probabilistic/randomized approach and the deterministic approach. The latter corresponds to time-bounded models (often called synchronous), where it is assumed that bounds on computational/communication delays are known a priori.

Within the framework of such models, many deterministic algorithms have been devised for solving all kinds of consensus problems (group membership, global time, etc.). Nevertheless, the validity of such algorithms is delimited by the validity the time-boundedness assumptions.

In special cases, the existence of bounds can be enforced or guaranteed by some specific hardware implementation. In the general case, however, possible violations of hypothesized bounds need to be detected at run time. The assumption coverage α depends on the magnitude of bounds and on the detection algorithms used. When taking the correct view that run-time assumption checking is required with time-bounded based designs meant to achieve some provable degree of predictability, one is led to realize that some theoretical results have limited scope of applicability.

A typical example is that of deterministic clock synchronization. Under the assumption that upper (U) and lower (L) bounds on communication delays are known, a number of optimal clock synchronization algorithms have been devised, that minimize errors experienced in performing remote clock reads by minimizing errors in estimating actual communication delays. However, in many cases, the only means whereby timing failures can be detected is by measuring actual communication delays, i.e. by measuring the difference between receive time and send time for every message, using local clocks. Unfortunately, correct detection of timing failures under this approach boils down to assuming that the condition $U-L << U$ (or, similarly, $U-L << L$) holds.

This severely diminishes the value of those optimal synchronization algorithms as, quite obviously, under such a condition, the initial problem (i.e. accurate remote clock read) vanishes.

A third approach to the problem of achieving time-bounded distributed consensus consists in adopting time-unbounded models and supplementing them with some minimal additional knowledge sufficient to break up established impossibility results[21]. Corresponding assumptions should yield values of α greater than those achievable with time-bounded models.

5. CONCLUSION

Real-time dependable distributed computing systems present challenges in principle and in practice. Related concepts and techniques are beginning to coalesce from the past decade of theoretical and experimental research.

This article addresses some of the important design issues in the area. A good question to ask is how will appropriate solutions be translated into commercial offerings.

Various communities of users would be interested in such offerings. However, at present time, there is simply no hardware product, no software product -- such as an operating system -- that would free users from having to deal with design issues such as those presented in this paper. Therefore, it is highly likely that appropriate solutions will appear first in distributed application run-time environments, sitting in between operating

systems and application-level software, until some leading companies move these solutions into operating systems and move basic mechanisms supporting these solutions into hardware.

REFERENCES

1 Thomas R.H., "A majority consensus approach to concurrency control for multiple copy databases", ACM Trans. on Database Systems, 4(2), June 1979, 180-209.

2 Lamport L., "Time, clocks and the ordering of events in a distributed system", Com. ACM, 21(7), July 1978, 558-565.

3 Le Lann G., "Distributed systems - Towards a formal approach", IFIP Congress, Toronto, 1977, North-Holland Pub., 155-160.

4 Jensen E.D., Locke C.L., Tokuda H., "A time-driven scheduling model for real-time operating systems", systems", IEEE Real-Time Systems Symp., 1985, 112-122.

5 Laprie J.C., "Dependability : a unified concept for reliable computing and fault-tolerance", Resilient Computer Systems, ed. T. Anderson, Collins and Wiley, 1989, 1-28.

6 Liu C.L., Layland J.W., "Scheduling algorithms for multiprogramming in a hard real-time environment", J.ACM, 20(1), January 1973, 46-61.

7 Sprunt B., Lehoczky J.P., Sha L., "Aperiodic task scheduling for hard real-time systems", The Journal of Real-Time Systems, 1, 1989, 27-69.

8 Flajolet P., Jacquet P., "Analytical model for tree communication protocols", NATO Advanced Study Institute on Flow Control of Congested Networks, Capri, Springer-Verlag, 1987.

9 Gallager R.G., "A perspective on multiaccess channels", IEEE Trans. on Information Theory, IT 31, 1985, 124-142.

10 Papadimitriou C.H., "Serializability of concurrent database updates", J.ACM, 26(4), October 1979, 631-653.

11 Eswaran K.P., et al., "The notions of consistency and predicate locks in a database system", Com. ACM, 19(11), November 1976, 624-633.

12 Bernstein P.A., Goodman N., "Concurrency control in distributed database systems", ACM Computing Surveys, 13(2), June 1981, 185-221.

13 Reed D.P., "Implementing atomic actions on decentralized data", ACM Trans. on Computer Systems, 1(1), February 1983, 3-23.

14 Le Lann G., "Algorithms for distributed data-sharing systems which use tickets", 3rd ACM/IEEE Berkeley Workshop on Distributed Databases and Computer Networks, August 1978, 259-272.

15 Gifford D.K., "Weighted voting for replicated data", 7th ACM SIGOPS Symp. on Operating Systems Principles, Pacific Grove, December 1979, 150-159.

16 Bernstein P.A., Hadzilacos V., Goodman N., "Concurrency control and recovery in database systems", Addison-Wesley Pub., ISBN 0-201-10715-5, 1987, 370 p.

17 Downing A.R., Greenberg I.B., Lunt T.F., "Issues in Distributed database security", Fifth Aerospace Computer Security Application Conference, 1989, 196-203.

18 Maimone W.T., Greenberg I.B., "Single-level multiversion schedulers for multilevel secure database systems", Sixth Annual Computer Security Applications Conference, 1990, 11 p.

19 Cristian F., "Understanding fault-tolerant distributed systems", Com. ACM, 34(2), February 1991, 56-78.

20 Fischer M.J., Lynch N.A., Paterson M.S., "Impossibility of distributed consensus with one faulty process", J.ACM, 32(2), April 1985, 374-382.

21 Chandra T.D., Toueg S., "Failure detectors for asynchronous systems", Symp. on Principles of Distributed Computing, Montreal, 1991.

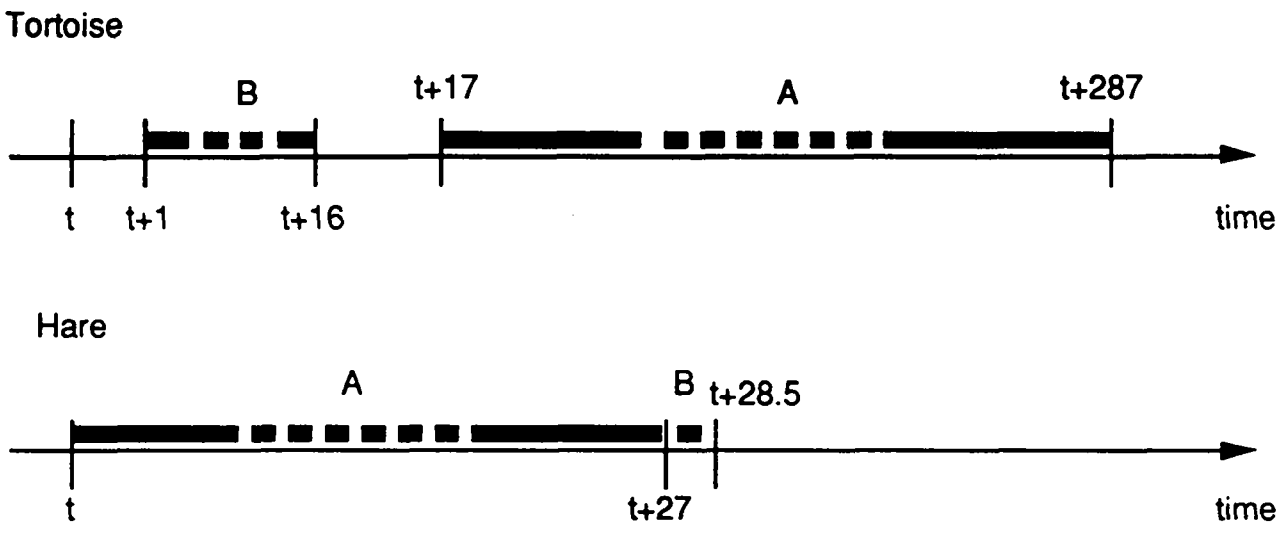
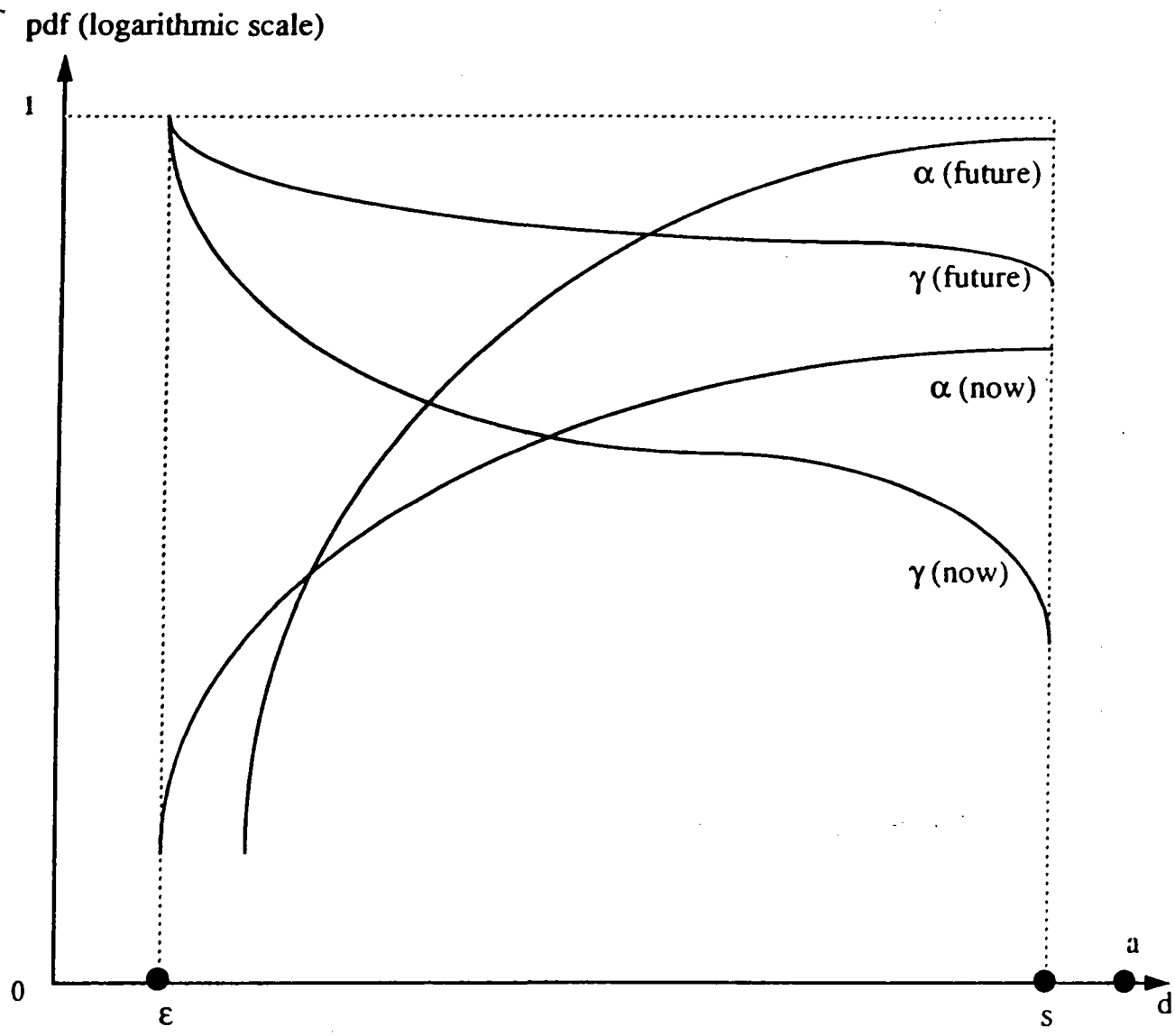


Figure 1 : A simple scheduling scenario



a = application intrinsic complexity

d = specification/design assumptions complexity (ranges between ε and s)

α = assumption coverage

γ= verification/proof coverage

Predictability P = αγ

Figure 2 : Anticipated evolution of predictability for ε-complex (minor complexity) and s-complex (satisfactory complexity) designs

Imprimé en France
par
. l'Institut National de Recherche en Informatique et en Automatique

ISSN 0249-6399